chapter

9

TRAP Routines and Subroutines

9.1 LC-3 TRAP Routines

9.1.1 Introduction

Recall Figure 8.5 of the previous chapter. In order to have the program successfully obtain input from the keyboard, it was necessary for the programmer (in Chapter 8) to know several things:

1. The hardware data registers for both the keyboard and the monitor: the monitor so a prompt could be displayed, and the keyboard so the program would know where to look for the input character.
2. The hardware status registers for both the keyboard and the monitor: the monitor so the program would know when it was OK to display the next character in the input prompt, and the keyboard so the program would know when someone had struck a key.
3. The asynchronous nature of keyboard input relative to the executing program.

This is beyond the knowledge of most application programmers. In fact, in the real world, if application programmers (or user programmers, as they are sometimes called) had to understand I/O at this level, there would be much less I/O and far fewer programmers in the business.

There is another problem with allowing user programs to perform I/O activity by directly accessing KBDR and KBSR. I/O activity involves the use of device registers that are shared by many programs. This means that if a user programmer

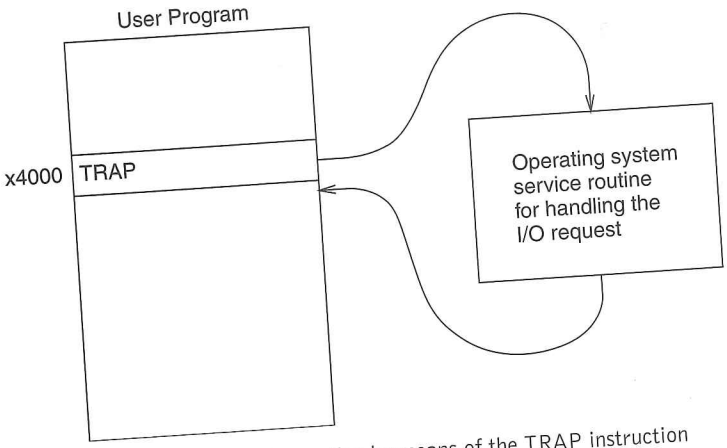


Figure 9.1 Invoking an OS service routine by means of the TRAP instruction

were allowed to access the hardware registers, and he/she messed up, it could create havoc for other user programs. Thus, it is ill-advised to give user programmers access to these registers. We say the hardware registers are **privileged** and accessible only to programs that have the proper degree of *privilege*.

The notion of privilege introduces a pretty big can of worms. Unfortunately, we cannot do much more than mention it here and leave serious treatment for later. For now, we simply note that there are resources that are not accessible to the user program, and access to those resources is controlled by endowing some programs with sufficient privilege and other programs without. Having said that, we move on to our problem at hand, a "better" solution for user programs that require input and/or output.

The simpler solution as well as the safer solution to the problem of user programs requiring I/O involves the TRAP instruction and the operating system. The operating system does have the proper degree of privilege.

We were introduced to the TRAP instruction in Chapter 5. We saw that for certain tasks, a user program could get the operating system to do the job for it by invoking the TRAP instruction. That way, the user programmer does not have to know the gory details previously mentioned, and other user programs are protected from the consequences of inept user programmers.

Figure 9.1 shows a user program that, upon reaching location x4000, needs an I/O task performed. The user program requests the operating system to perform the task on behalf of the user program. The operating system takes control of the computer, handles the request specified by the TRAP instruction, and then returns control to the user program, at location x4001. We often refer to the request made by the user program as a *service call* or a *system call*.

9.1.2 The TRAP Mechanism

The TRAP mechanism involves several elements, as follows:

1. **A set of service routines** executed on behalf of user programs by the operating system. These are part of the operating system and start at

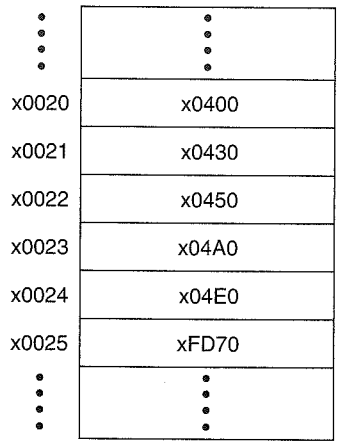


Figure 9.2 The Trap Vector Table

arbitrary addresses in memory. The LC-3 was designed so that up to 256 service routines can be specified. Table A.2 in Appendix A contains the LC-3's current complete list of operating system service routines.

2. **A table of the starting addresses** of these 256 service routines. This table is stored in memory locations x0000 to x00FF. The table is referred to by various names by various companies. One company calls this table the System Control Block. Another company calls it the Trap Vector Table. Figure 9.2 provides a snapshot of the Trap Vector Table of the LC-3, with specific starting addresses highlighted. Among the starting addresses are the one for the character output service routine (location x0430), which is contained in location x0021, the one for the keyboard input service routine (location x04A0), contained in location x0023, and the one for the machine halt service routine (location xFD70), contained in location x0025.
3. **The TRAP instruction.** When a user program wishes to have the operating system execute a specific service routine on behalf of the user program, and then return control to the user program, the user program uses the TRAP instruction.
4. A **linkage** back to the user program. The service routine must have a mechanism for returning control to the user program.

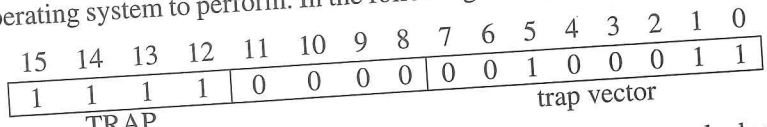
9.1.3 The TRAP Instruction

The TRAP instruction causes the service routine to execute by doing two things:

- It changes the PC to the starting address of the relevant service routine on the basis of its trap vector.
- It provides a way to get back to the program that initiated the TRAP instruction. The "way back" is referred to as a *linkage*.

The TRAP instruction is specified as follows. The **TRAP** instruction is made up of two parts: the TRAP opcode 1111 and the trap vector (bits [7:0]). Bits [11:8]

must be zero. The trap vector identifies the service routine the user program wants the operating system to perform. In the following example, the trap vector is x23.



The EXECUTE phase of the TRAP instruction's instruction cycle does four things:

1. The 8-bit trap vector is zero-extended to 16 bits to form an address, which is loaded into the MAR. For the trap vector x23, that address is x0023, which is the address of an entry in the Trap Vector Table.
2. The Trap Vector Table is in memory locations x0000 to x00FF. The entry at x0023 is read and its contents, in this case x04A0 (see Figure 9.2), are loaded into the MDR.
3. The general purpose register R7 is loaded with the current contents of the PC. This will provide a way back to the user program, as will become clear momentarily.
4. The contents of the MDR are loaded into the PC, completing the instruction cycle.

Since the PC now contains x04A0, processing continues at memory address x04A0.

Location x04A0 is the starting address of the operating system service routine to input a character from the keyboard. We say the trap vector "points" to the starting address of the TRAP routine. Thus, TRAP x23 causes the operating system to start executing the keyboard input service routine.

In order to return to the instruction following the TRAP instruction in the user program (after the service routine has ended), there must be some mechanism for saving the address of the user program's next instruction. Step 3 of the EXECUTE phase listed above provides this linkage. By storing the PC in R7 before loading the PC with the starting address of the service routine, the TRAP instruction provides the service routine with all the information it needs to return control to the user program at the proper location. You know that the PC was already updated (in the FETCH phase of the TRAP instruction) to point to the next instruction. Thus, at the start of execution of the trap service routine, R7 contains the address of the instruction in the user program that follows the TRAP instruction.

9.1.4 The Complete Mechanism

We have shown in detail how the TRAP instruction invokes the service routine to do the user program's bidding. We have also shown how the TRAP instruction provides the information that the service routine needs to return control to the correct place in the user program. The only thing left is to show the actual instruction in the service routine that returns control to the correct place in the user program. Recall the JMP instruction from Chapter 5. Assume that during the execution of the trap service routine, the contents of R7 was not changed. If

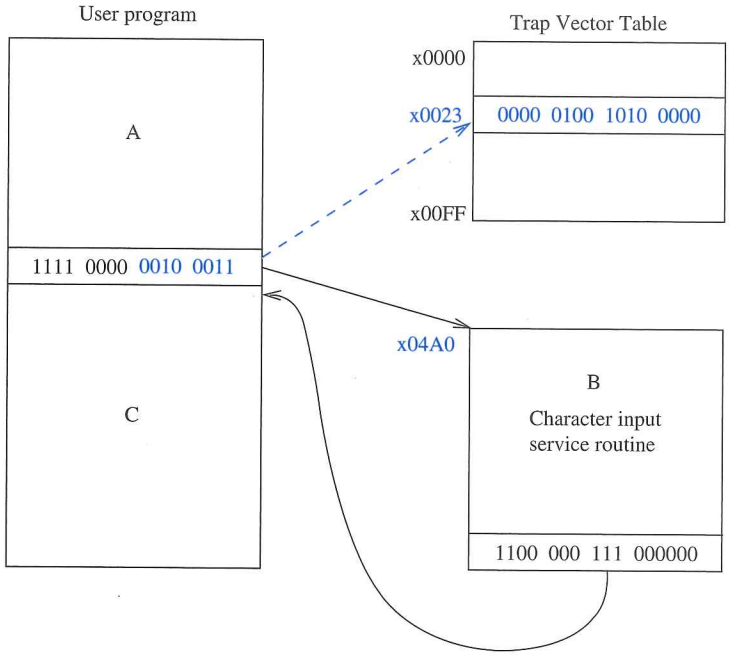


Figure 9.3 Flow of control from a user program to an OS service routine and back

that is the case, control can return to the correct location in the user program by executing JMP R7 as the last instruction in the trap service routine.

Figure 9.3 shows the LC-3 using the TRAP instruction and the JMP instruction to implement the example of Figure 9.1. The flow of control goes from (A) within a user program that needs a character input from the keyboard, to (B) the operating system service routine that performs that task on behalf of the user program, back to the user program (C) that presumably uses the information contained in the input character.

Recall that the computer continually executes its instruction cycle (FETCH, DECODE, etc.). As you know, the way to change the flow of control is to change the contents of the PC during the EXECUTE phase of the current instruction. In that way, the next FETCH will be at a redirected address.

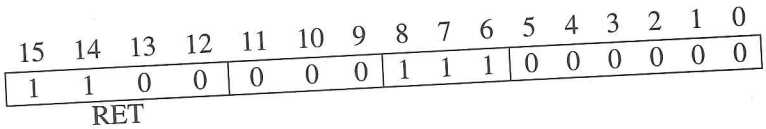
Thus, to request the character input service routine, we use the TRAP instruction with trap vector x23 in our user program. Execution of that instruction causes the contents of memory location x0023 (which, in this case, contains x04A0) to be loaded into the PC and the address of the instruction following the TRAP instruction to be loaded into R7. The dashed lines on Figure 9.3 show the use of the trap vector to obtain the starting address of the trap service routine from the Trap Vector Table.

The next instruction cycle starts with the FETCH of the contents of x04A0, which is the first instruction of the operating system service routine that requests (and accepts) keyboard input. That service routine, as we will see momentarily, is patterned after the keyboard input routine we studied in Section 8.4. Recall that

upon completion of that input routine (see Figure 8.5), R0 contains the ASCII code of the key that was typed.

The trap service routine executes to completion, ending with the JMP R7 instruction. Execution of JMP R7 loads the PC with the contents of R7. If R7 was not changed during execution of the service routine, it still contains the address of the instruction following the TRAP instruction in the initiating user program. Thus, the user program resumes execution, with R0 containing the ASCII code of the keyboard character that was typed.

The JMP R7 instruction is so convenient for providing a return to the user program that the LC-3 assembly language provides the mnemonic RET for this instruction, as follows:



The following program is provided to illustrate the use of the TRAP instruction. It can also be used to amuse the average four-year-old!

Example 9.1

Write a game program to do the following: A person is sitting at a keyboard. Each time the person types a capital letter, the program outputs the lowercase version of that letter. If the person types a 7, the program terminates.

The following LC-3 assembly language program will do the job.

```
01              .ORIG x3000
02              LD    R2,TERM   ; Load -7
03              LD    R3,ASCII  ; Load ASCII difference
04    AGAIN     TRAP  x23       ; Request keyboard input
05              ADD   R1,R2,R0  ; Test for terminating
06              BRz   EXIT      ; character
07              ADD   R0,R0,R3  ; Change to lowercase
08              TRAP  x21       ; Output to the monitor
09              BRnzp AGAIN     ; ... and do it again!
0A    TERM      .FILL xFFC9     ; FFC9 is negative of ASCII 7
0B    ASCII     .FILL x0020
0C    EXIT      TRAP  x25       ; Halt
0D              .END
```

The program executes as follows: The program first loads constants xFFC9 and x0020 into R2 and R3. The constant xFFC9, which is the negative of the ASCII code for 7, is used to test the character typed at the keyboard to see if the four-year-old wants to continue playing. The constant x0020 is the zero-extended difference between the ASCII code for a capital letter and the ASCII code for that same letter's lowercase representation. For example, the ASCII code for **A** is x41; the ASCII code for **a** is x61. The ASCII codes for **Z** and **z** are x5A and x7A, respectively.

Then TRAP x23 is executed, which invokes the keyboard input service routine. When the service routine is finished, control returns to the application program (at line 05), and R0 contains the ASCII code of the character typed. The ADD and BRz instructions test for the terminating character 7. If the character typed is not a 7, the ASCII uppercase/lowercase difference (x0020) is added to the input ASCII code, storing the result in R0. Then a TRAP to the monitor output service routine is called. This causes the lowercase representation of the same letter to be displayed on the monitor. When control returns to the application program (this time at line 09), an unconditional BR to AGAIN is executed, and another request for keyboard input appears.

The correct operation of the program in this example assumes that the person sitting at the keyboard only types capital letters and the value 7. What if the person types a $? A better solution to Example 9.1 would be a program that tests the character typed to be sure it really is a capital letter from among the 26 capital letters in the alphabet, and if it is not, takes corrective action.

Question: Augment this program to add the test for bad data. That is, write a program that will type the lowercase representation of any capital letter typed and will terminate if anything other than a capital letter is typed. See Exercise 9.6.

9.1.5 TRAP Routines for Handling I/O

With the constructs just provided, the input routine described in Figure 8.5 can be slightly modified to be the input service routine shown in Figure 9.4. Two changes are needed: (1) We add the appropriate .ORIG and .END pseudo-ops. .ORIG specifies the starting address of the input service routine—the address found at location x0023 in the Trap Vector Table. And (2) we terminate the input service routine with the JMP R7 instruction (mnemonically, RET) rather than the BR NEXT_TASK, as is done on line 20 in Figure 8.5. We use JMP R7 because the service routine is invoked by TRAP x23. It is not part of the user program, as was the case in Figure 8.5.

The output routine of Section 8.3.2 can be modified in a similar way, as shown in Figure 9.5. The results are input (Figure 9.4) and output (Figure 9.5) service routines that can be invoked simply and safely by the TRAP instruction with the appropriate trap vector. In the case of input, upon completion of TRAP x23, R0 contains the ASCII code of the keyboard character typed. In the case of output, the initiating program must load R0 with the ASCII code of the character it wishes displayed on the monitor and then invoke TRAP x21.

9.1.6 TRAP Routine for Halting the Computer

Recall from Section 4.5 that the RUN latch is ANDed with the crystal oscillator to produce the clock that controls the operation of the computer. We noted that if that 1-bit latch was cleared, the output of the AND gate would be 0, stopping the clock.

Years ago, most ISAs had a HALT instruction for stopping the clock. Given how infrequently that instruction is executed, it seems wasteful to devote an opcode to it. In many modern computers, the RUN latch is cleared by a TRAP

```
01      ;  Service Routine for Keyboard Input
02      ;
03              .ORIG   x04A0
04      START   ST      R1,SaveR1       ; Save the values in the registers
05              ST      R2,SaveR2       ; that are used so that they
06              ST      R3,SaveR3       ; can be restored before RET
07      ;
08              LD      R2,Newline
09      L1      LDI     R3,DSR          ; Check DDR --  is it free?
0A              BRzp    L1
0B              STI     R2,DDR          ; Move cursor to new clean line
0C      ;
0D              LEA     R1,Prompt       ; Prompt is starting address
0E                                      ; of prompt string
1F      Loop    LDR     R0,R1,#0        ; Get next prompt character
10              BRz     Input           ; Check for end of prompt string
11      L2      LDI     R3,DSR
12              BRzp    L2
13              STI     R0,DDR          ; Write next character of
14                                      ; prompt string
15              ADD     R1,R1,#1        ; Increment prompt pointer
16              BRnzp   Loop
17      ;
18      Input   LDI     R3,KBSR         ; Has a character been typed?
19              BRzp    Input
1A              LDI     R0,KBDR         ; Load it into R0
1B      L3      LDI     R3,DSR
1C              BRzp    L3
1D              STI     R0,DDR          ; Echo input character
1E                                      ; to the monitor
1F      ;
20      L4      LDI     R3,DSR
21              BRzp    L4
22              STI     R2,DDR          ; Move cursor to new clean line
23              LD      R1,SaveR1       ; Service routine done, restore
24              LD      R2,SaveR2       ; original values in registers.
25              LD      R3,SaveR3
26              RET                     ; Return from trap (i.e., JMP R7)
27      ;
28      SaveR1  .BLKW   1
29      SaveR2  .BLKW   1
2A      SaveR3  .BLKW   1
2B      DSR     .FILL   xFE04
2C      DDR     .FILL   xFE06
2D      KBSR    .FILL   xFE00
2E      KBDR    .FILL   xFE02
2F      Newline .FILL   x000A           ; ASCII code for newline
30      Prompt  .STRINGZ "Input a character>"
31              .END
```

Figure 9.4 Character input service routine

```
01              .ORIG   x0430          ; System call starting address
02              ST      R1, SaveR1     ; R1 will be used to poll the DSR
03                                     ; hardware
04    ; Write the character
05    TryWrite  LDI     R1, DSR        ; Get status
06              BRzp    TryWrite       ; Bit 15 on says display is ready
07    WriteIt   STI     R0, DDR        ; Write character
08
09    ; return from trap
0A    Return    LD      R1, SaveR1     ; Restore registers
0B              RET                    ; Return from trap (JMP R7, actually)
0C    DSR       .FILL   xFE04          ; Address of display status register
0D    DDR       .FILL   xFE06          ; Address of display data register
0E    SaveR1    .BLKW   1
0F              .END
```

Figure 9.5 Character output service routine

routine. In the LC-3, the RUN latch is bit [15] of the Machine Control Register, which is memory-mapped to location xFFFE. Figure 9.6 shows the trap service routine for halting the processor, that is, for stopping the clock.

First (lines 02, 03, and 04), registers R7, R1, and R0 are saved. R1 and R0 are saved because they are needed by the service routine. R7 is saved because its contents will be overwritten after TRAP x21 executes (line 09). Then (lines 08 through 0D), the banner *Halting the machine* is displayed on the monitor. Finally (lines 11 through 14), the RUN latch (MCR[15]) is cleared by ANDing the MCR with 0111111111111111. That is, MCR[14:0] remains unchanged, but MCR[15] is cleared. *Question*: What instruction (or trap service routine) can be used to start the clock?

```
01                .ORIG   xFD70      ; Where this routine resides
02              ST      R7, SaveR7
03              ST      R1, SaveR1 ; R1: a temp for MC register
04              ST      R0, SaveR0 ; R0 is used as working space
05
06  ; print message that machine is halting
07
08              LD      R0, ASCIINewLine
09              TRAP    x21
0A              LEA     R0, Message
0B              TRAP    x22
0C              LD      R0, ASCIINewLine
0D              TRAP    x21
0E  ;
0F  ; clear bit 15 at xFFFE to stop the machine
10  ;
11              LDI     R1, MCR     ; Load MC register into R1
12              LD      R0, MASK    ; R0 = x7FFF
13              AND     R0, R1, R0 ; Mask to clear the top bit
14              STI     R0, MCR     ; Store R0 into MC register
15  ;
```

Figure 9.6 HALT service routine for the LC-3

```
16  ; return from HALT routine.
17  ; (how can this routine return if the machine is halted above?)
18  ;
19                  LD      R1, SaveR1 ; Restore registers
1A                  LD      R0, SaveR0
1B                  LD      R7, SaveR7
1C                  RET                ; JMP R7, actually
1D  ;
1E  ; Some constants
1F  ;
20  ASCIINewLine    .FILL   x000A
21  SaveR0          .BLKW   1
22  SaveR1          .BLKW   1
23  SaveR7          .BLKW   1
24  Message         .STRINGZ   "Halting the machine."
25  MCR             .FILL   xFFFE      ; Address of MCR
26  MASK            .FILL   x7FFF      ; Mask to clear the top bit
27                  .END
```

Figure 9.6 HALT service routine for the LC-3 (continued)

9.1.7 Saving and Restoring Registers

One item we have mentioned in passing that we should emphasize more explicitly is the need to save the value in a register

- if the value will be destroyed by some subsequent action, and
- if we will need to use it after that subsequent action.

Suppose we want to input from the keyboard 10 decimal digits, convert their ASCII codes into their binary representations, and store the binary values in 10 successive memory locations, starting at the address `Binary`. The following program fragment does the job.

```
01                  LEA     R3,Binary   ; Initialize to first location
02                  LD      R6,ASCII    ; Template for line 05
03                  LD      R7,COUNT    ; Initialize to 10
04  AGAIN           TRAP    x23         ; Get keyboard input
05                  ADD     R0,R0,R6    ; Strip ASCII template
06                  STR     R0,R3,#0    ; Store binary digit
07                  ADD     R3,R3,#1    ; Increment pointer
08                  ADD     R7,R7,#-1   ; Decrement COUNT.
09                  BRp     AGAIN       ; More characters?
0A                  BRnzp   NEXT_TASK   ;
0B  ASCII           .FILL   xFFD0       ; Negative of x0030.
0C  COUNT           .FILL   #10
0D  Binary          .BLKW   #10
```

The first step in the program fragment is initialization. We load R3 with the starting address of the memory space set aside to store the 10 decimal digits. We load R6 with the negative of the ASCII template. This is used to subtract x0030 from each ASCII code. We load R7 with 10, the initial value of the count. Then we execute the loop 10 times, each time getting a character from the keyboard, stripping away the ASCII template, storing the binary result, and testing to see if we are done. But the program does not work! Why? *Answer:* The TRAP instruction in line 04 replaces the value 10 that was loaded into R7 in line 03 with the address of the ADD R0,R0,R6 instruction. Therefore, the instructions in lines 08 and 09 do not perform the loop control function they were programmed to do.

The message is this: If a value in a register will be needed after something else is stored in that register, we must *save* it before the something else happens and *restore* it before we can subsequently use it. We save a register value by storing it in memory; we restore it by loading it back into the register. In Figure 9.6, line 03 contains the ST instruction that saves R1, line 11 contains the LDI instruction that loads R1 with a value to do the work of the trap service routine, line 19 contains the LD instruction that restores R1 to its original value before the service routine was called, and line 22 sets aside a location in memory for storing R1.

The save/restore problem can be handled either by the initiating program before the TRAP occurs or by the called program (for example, the service routine) after the TRAP instruction executes. We will see in Section 9.2 that the same problem exists for another class of calling/called programs, the subroutine mechanism.

We use the term *caller-save* if the calling program handles the problem. We use the term *callee-save* if the called program handles the problem. The appropriate one to handle the problem is the one that knows which registers will be destroyed by subsequent actions.

The callee knows which registers it needs to do the job of the called program. Therefore, before it starts, it saves those registers with a sequence of stores. After it finishes, it restores those registers with a sequence of loads. And it sets aside memory locations to save those register values. In Figure 9.6, the HALT routine needs R0 and R1. So it saves their values with ST instructions in lines 03 and 04, restores their values with LD instructions in lines 19 and 1A, and sets aside memory locations for these values in lines 21 and 22.

The caller knows what damage will be done by instructions under its control. Again, in Figure 9.6, the caller knows that each instance of the TRAP instruction will destroy what is in R7. So, before the first TRAP instruction in the HALT service routine is executed, R7 is saved. After the last TRAP instruction in the HALT service routine is executed, R7 is restored.

9.2 Subroutines

We have just seen how programmers' productivity can be enhanced if they do not have to learn details of the I/O hardware, but can rely instead on the operating system to supply the program fragments needed to perform those tasks. We also mentioned in passing that it is kind of nice to have the operating system access these device registers so we do not have to be at the mercy of some other user programmer.

We have seen that a request for a service routine is invoked in the user program by the TRAP instruction and handled by the operating system. Return to the initiating program is obtained via the JMP R7 instruction.

In a similar vein, it is often useful to be able to invoke a program fragment multiple times within the same program without having to specify its details all over again in the source program each time it is needed. In addition, it is sometimes the case that one person writes a program that requires such fragments and another person writes the fragments.

Also, one might require a fragment that has been supplied by the manufacturer or by some independent software supplier. It is almost always the case that collections of such fragments are available to user programmers to free them from having to write their own. These collections are referred to as *libraries*. An example is the Math Library, which consists of fragments that execute such functions as **square root, sine,** and **arctangent.**

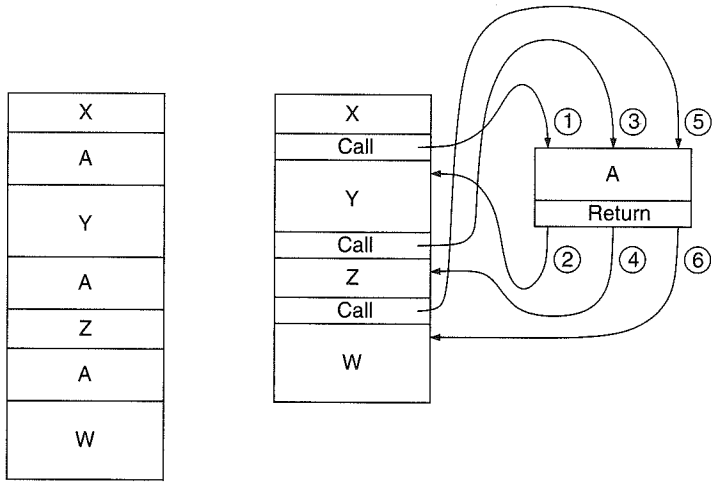
For all of these reasons, it is good to have a way to use program fragments efficiently. Such program fragments are called *subroutines,* or alternatively, *procedures*, or in C terminology, *functions*. The mechanism for using them is referred to as a *Call/Return mechanism.*

9.2.1 The Call/Return Mechanism

Figure 9.4 provides a simple illustration of a fragment that must be executed multiple times within the same program. Note the three instructions starting at symbolic address L1. Note also the three instructions starting at addresses L2, L3, and L4. Each of these four 3-instruction sequences do the following:

```
LABEL   LDI     R3,DSR
BRzp    LABEL
STI     Reg,DDR
```

Two of the four program fragments store the contents of R0 and the other two store the contents of R2, but that is easy to take care of, as we will see. The main point is that, aside from the small nuisance of which register is being used for the source for the STI instruction, the four program fragments do exactly the same thing. The Call/Return mechanism allows us to execute this one 3-instruction sequence multiple times while requiring us to include it as a subroutine in our program only once.



(a) Without subroutines (b) With subroutines

Figure 9.7 Instruction execution flow with/without subroutines

The call mechanism computes the starting address of the subroutine, loads it into the PC, and saves the return address for getting back to the next instruction in the calling program. The return mechanism loads the PC with the return address. Figure 9.7 shows the instruction execution flow for a program with and without subroutines.

The Call/Return mechanism acts very much like the TRAP instruction in that it redirects control to a program fragment while saving the linkage back to the calling program. In both cases, the PC is loaded with the starting address of the program fragment, while R7 is loaded with the address that is needed to get back to the calling program. The last instruction in the program fragment, whether the fragment is a trap service routine or a subroutine, is the JMP R7 instruction, which loads the PC with the contents of R7, thereby returning control to the instruction following the calling instruction.

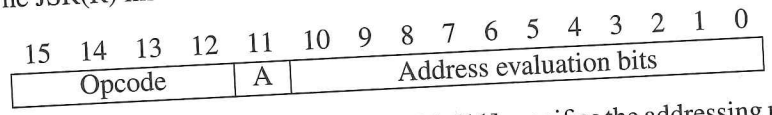
There is an important difference between subroutines and the service routines that are called by the TRAP instruction. Although it is somewhat beyond the scope of this course, we will mention it briefly. It has to do with the nature of the work that the program fragment is being asked to do. In the case of the TRAP instruction (as we saw), the service routines involve operating system resources, and they generally require privileged access to the underlying hardware of the computer. They are written by systems programmers charged with managing the resources of the computer. In the case of subroutines, they are either written by the same programmer who wrote the program containing the calling instruction, or they are written by a colleague, or they are provided as part of a library. In all cases, they involve resources that cannot mess up other people's programs, and so we are not concerned that they are part of a user program.

9.2.2 The JSR(R) Instruction

The LC-3 specifies one opcode for calling subroutines, **0100.** The instruction uses one of two addressing modes for computing the starting address of the subroutine, PC-relative addressing or Base addressing. The LC-3 assembly language provides two different mnemonic names for the opcode, JSR and JSRR, depending on which addressing mode is used.

The instruction does two things. It saves the return address in R7 and it computes the starting address of the subroutine and loads it into the PC. The return address is the incremented PC, which points to the instruction following the JSR or JSRR instruction in the calling program.

The JSR(R) instruction consists of three parts.

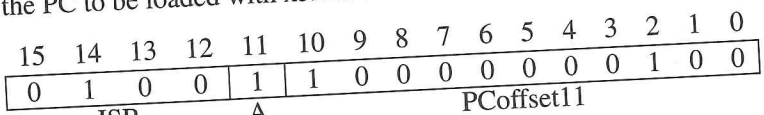


Bits [15:12] contain the opcode, 0100. Bit [11] specifies the addressing mode, the value 1 if the addressing mode is PC-relative, and the value 0 if the addressing mode is Base addressing. Bits [10:0] contain information that is used to evaluate the starting address of the subroutine. The only difference between JSR and JSRR is the addressing mode that is used for evaluating the starting address of the subroutine.

JSR

The **JSR** instruction computes the target address of the subroutine by sign-extending the 11-bit offset (bits [10:0]) of the instruction to 16 bits and adding that to the incremented PC. This addressing mode is almost identical to the addressing mode of the LD and ST instructions, except 11 bits of PCoffset are used, rather than nine bits as is the case for LD and ST.

If the following JSR instruction is stored in location x4200, its execution will cause the PC to be loaded with x3E05 and R7 to be loaded with x4201.

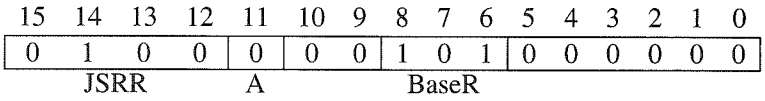


JSRR

The JSRR instruction is exactly like the JSR instruction except for the addressing mode. **JSRR** obtains the starting address of the subroutine in exactly the same way the JMP instruction does, that is, it uses the contents of the register specified by bits [8:6] of the instruction.

If the following JSRR instruction is stored in location x420A, and if R5 contains x3002, the execution of the JSRR will cause R7 to be loaded with x420B, and the PC to be loaded with x3002.

Question: What important feature does the JSRR instruction provide that the JSR instruction does not provide?



9.2.3 The TRAP Routine for Character Input, Revisited

Let's look again at the keyboard input service routine of Figure 9.4. In particular, let's look at the three-line sequence that occurs at symbolic addresses L1, L2, L3, and L4:

```
LABEL   LDI     R3,DSR
        BRzp    LABEL
        STI     Reg,DDR
```

Can the JSR/RET mechanism enable us to replace these four occurrences of the same sequence with a single subroutine? *Answer:* Yes, **almost.**

Figure 9.8, our "improved" keyboard input service routine, contains

```
        JSR     WriteChar
```

at lines 05, 0B, 11, and 14, and the four-instruction subroutine

```
WriteChar   LDI     R3,DSR
            BRzp    WriteChar
            STI     R2,DDR
            RET
```

at lines 1D through 20. Note the RET instruction (actually, JMP R7) that is needed to terminate the subroutine.

Note the hedging: *almost.* In the original sequences starting at L2 and L3, the STI instruction forwards the contents of R0 (not R2) to the DDR. We can fix that easily enough, as follows: In line 09 of Figure 9.8, we use

```
        LDR     R2,R1,#0
```

instead of

```
        LDR     R0,R1,#0
```

This causes each character in the prompt to be loaded into R2. The subroutine Writechar forwards each character from R2 to the DDR.

In line 10 of Figure 9.8, we insert the instruction

```
        ADD     R2,R0,#0
```

in order to move the keyboard input (which is in R0) into R2. The subroutine Writechar forwards it from R2 to the DDR. Note that R0 still contains the keyboard input. Furthermore, since no subsequent instruction in the service routine loads R0, R0 still contains the keyboard input after control returns to the user program.

In line 13 of Figure 9.8, we insert the instruction

```
        LD      R2,Newline
```

in order to move the "newline" character into R2. The subroutine Writechar forwards it from R2 to the DDR.

Finally, we note that unlike Figure 9.4, this trap service routine contains several instances of the JSR instruction. Thus any linkage back to the calling

```
01                      .ORIG   x04A0
02   START              ST      R7,SaveR7
03                      JSR     SaveReg
04                      LD      R2,Newline
05                      JSR     WriteChar
06                      LEA     R1,PROMPT
07   ;
08   ;
09   Loop               LDR     R2,R1,#0     ; Get next prompt char
0A                      BRz     Input
0B                      JSR     WriteChar
0C                      ADD     R1,R1,#1
0D                      BRnzp   Loop
0E   ;
0F   Input              JSR     ReadChar
10                      ADD     R2,R0,#0     ; Move char to R2 for writing
11                      JSR     WriteChar    ; Echo to monitor
12   ;
13                      LD      R2, Newline
14                      JSR     WriteChar
15                      JSR     RestoreReg
16                      LD      R7,SaveR7
17                      RET                  ; JMP R7 terminates
                                               the TRAP routine
18   ;
19   SaveR7             .FILL   x0000
1A   Newline            .FILL   x000A
1B   Prompt             .STRINGZ  "Input a character>"
1C   ;
1D   WriteChar          LDI     R3,DSR
1E                      BRzp    WriteChar
1F                      STI     R2,DDR
20                      RET                  ; JMP R7 terminates subroutine
21   DSR                .FILL   xFE04
22   DDR                .FILL   xFE06
23    ;
24   ReadChar           LDI     R3,KBSR
25                      BRzp    ReadChar
26                      LDI     R0,KBDR
27                      RET
28   KBSR               .FILL   xFE00
29   KBDR               .FILL   xFE02
2A    ;
2B   SaveReg              ST      R1,SaveR1
2C                        ST      R2,SaveR2
2D                        ST      R3,SaveR3
2E                        ST      R4,SaveR4
2F                        ST      R5,SaveR5
30                        ST      R6,SaveR6
31                        RET
32   ;
33   RestoreReg           LD      R1,SaveR1
34                        LD      R2,SaveR2
35                        LD      R3,SaveR3
36                        LD      R4,SaveR4
37                        LD      R5,SaveR5
38                        LD      R6,SaveR6
39                        RET
3A   SaveR1               .FILL   x0000
3B   SaveR2               .FILL   x0000
3C   SaveR3               .FILL   x0000
3D   SaveR4               .FILL   x0000
3E   SaveR5               .FILL   x0000
3F   SaveR6               .FILL   x0000
40                        .END
```

Figure 9.8 The LC-3 trap service routine for character input

program that was contained in R7 when the service routine started execution was long ago overwritten (by the first JSR instruction, actually, in line 03). Therefore, we save R7 in line 02 before we execute our first JSR instruction, and we restore R7 in line 16 after we execute our last JSR instruction.

Figure 9.8 is the actual LC-3 trap service routine provided for keyboard input.

9.2.4 PUTS: Writing a Character String to the Monitor

Before we leave the example of Figure 9.8, note the code on lines 09 through 0D. This fragment of the service routine is used to write the sequence of characters *Input a character* to the monitor. A sequence of characters is often referred to as a *string of characters* or a *character string*. This fragment is also present in Figure 9.6, with the result that *Halting the machine* is written to the monitor. In fact, it is so often the case that a user program needs to write a string of characters to the monitor that this function is given its own trap vector in the LC-3 operating system. Thus, if a user program requires a character string to be written to the monitor, it need only provide (in R0) the starting address of the character string, and then invoke TRAP x22. In LC-3 assembly language this TRAP is called *PUTS*.

Thus, PUTS (or TRAP x22) causes control to be passed to the operating system, and the procedure shown in Figure 9.9 is executed. Note that PUTS is the code of lines 09 through 0D of Figure 9.8, with a few minor adjustments.

9.2.5 Library Routines

We noted early in this section that there are many uses for the Call/Return mechanism, among them the ability of a user program to call library subroutines that are usually delivered as part of the computer system. Libraries are provided as a convenience to the user programmer. They are legitimately advertised as "productivity enhancers" since they allow the user programmer to use them without having to know or learn much of their inner details. For example, a user programmer knows what a square root is (we abbreviate **SQRT**), and may need to use sqrt(x) for some value x but does not have a clue as to how to write a program to do it, and probably would rather not have to learn how.

A simple example illustrates the point. We have lost our key and need to get into our apartment. We can lean a ladder up against the wall so that the ladder touches the bottom of our open window, 24 feet above the ground. There is a 10-foot flower bed on the ground along the edge of the wall, so we need to keep the base of the ladder outside the flower bed. How big a ladder do we need so that we can lean it against the wall and climb through the window? Or, stated less colorfully: If the sides of a right triangle are 24 feet and 10 feet, how big is the hypotenuse (see Figure 9.10)?

We remember from high school that Pythagoras answered that one for us:

$$c^2 = a^2 + b^2$$

```
01      ; This service routine writes a NULL-terminated string to the console.
02      ; It services the PUTS service call (TRAP x22).
03      ; Inputs: R0 is a pointer to the string to print.
04      ;
05                  .ORIG   x0450           ; Where this ISR resides
06                  ST      R7, SaveR7      ; Save R7 for later return
07                  ST      R0, SaveR0      ; Save other registers that
08                  ST      R1, SaveR1      ; are needed by this routine
09                  ST      R3, SaveR3      ;
0A      ;
0B      ; Loop through each character in the array
0C      ;
0D      Loop        LDR     R1, R0, #0      ; Retrieve the character(s)
0E                  BRz     Return          ; If it is 0, done
0F      L2          LDI     R3,DSR
10                  BRzp    L2
11                  STI     R1, DDR         ; Write the character
12                  ADD     R0, R0, #1      ; Increment pointer
13                  BRnzp   Loop            ; Do it all over again
14      ;
15      ; Return from the request for service call
16      Return      LD      R3, SaveR3
17                  LD      R1, SaveR1
18                  LD      R0, SaveR0
19                  LD      R7, SaveR7
1A                  RET
1B      ;
1C      ; Register locations
1D      DSR         .FILL   xFE04
1E      DDR         .FILL   xFE06
1F      SaveR0      .FILL   x0000
20      SaveR1      .FILL   x0000
21      SaveR3      .FILL   x0000
22      SaveR7      .FILL   x0000
23                  .END
```

Figure 9.9 The LC-3 PUTS service routine

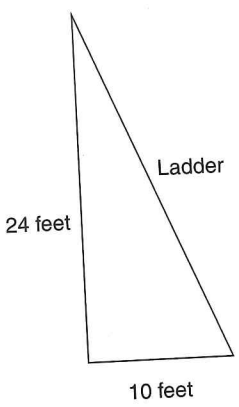


Figure 9.10 Solving for the length of the hypotenuse

Knowing a and b, we can easily solve for c by taking the square root of the sum of a^2 and b^2. Taking the sum is not hard—the LC-3 ADD instruction will do the job. The square is also not hard; we can multiply two numbers by a sequence of additions. But how does one get the square root? The structure of our solution is shown in Figure 9.11.

The subroutine SQRT has yet to be written. If it were not for the Math Library, the programmer would have to pick up a math book (or get someone to do it for him/her), check out the Newton-Raphson method, and produce the missing subroutine.

However, with the Math Library, the problem pretty much goes away. Since the Math Library supplies a number of subroutines (including SQRT), the user programmer can continue to be ignorant of the likes of Newton-Raphson. The user still needs to know the label of the target address of the library routine that performs the square root function, where to put the argument x, and where to expect the result SQRT(x). But these are easy conventions that can be obtained from the documentation associated with the Math Library.

```
01              ...
02              ...
03              LD      R0,SIDE1
04              BRz     S1
05              JSR     SQUARE
06      S1      ADD     R1,R0,#0
07              LD      R0,SIDE2
08              BRz     S2
09              JSR     SQUARE
0A      S2      ADD     R0,R0,R1
0B              JSR     SQRT
0C              ST      R0,HYPOT
0D              BRnzp   NEXT_TASK
0E      SQUARE  ADD     R2,R0,#0
0F              ADD     R3,R0,#0
10      AGAIN   ADD     R2,R2,#-1
11              BRz     DONE
12              ADD     R0,R0,R3
13              BRnzp   AGAIN
14      DONE    RET
15      SQRT    ...             ; R0 <-- SQRT(R0)
16              ...             ;
17              ...             ; How do we write this subroutine?
18              ...             ;
19              ...             ;
1A              RET
1B      SIDE1   .BLKW   1
1C      SIDE2   .BLKW   1
1D      HYPOT   .BLKW   1
1E              ...
1F              ...
```

Figure 9.11 A program fragment to compute the hypotenuse of a right triangle

If the library routine starts at address SQRT, and the argument is provided to the library routine at R0, and the result is obtained from the library routine at R0, Figure 9.11 reduces to Figure 9.12.

Two things are worth noting:

- *Thing 1*—The programmer no longer has to worry about how to compute the square root function. The library routine does that for us.
- *Thing 2*—The pseudo-op .EXTERNAL. We already saw in Section 7.4.2 that this pseudo-op tells the assembler that the label (SQRT), which is needed to assemble the .FILL pseudo-op in line 19, will be supplied by some other program fragment (i.e., module) and will be combined with this program fragment (i.e., module) when the *executable image* is produced. The executable image is the binary module that actually executes. The executable image is produced at *link* time.

This notion of combining multiple modules at link time to produce an executable image is the normal case. Figure 9.13 illustrates the process. You will see concrete examples of this when we work with the programming language C in the second half of this course.

```
01                ...
02                ...
03                .EXTERNAL SQRT
04                ...
05                ...
06                LD      R0,SIDE1
07                BRz     1$
08                JSR     SQUARE
09      1$        ADD     R1,R0,#0
0A                LD      R0,SIDE2
0B                BRz     2$
0C                JSR     SQUARE
0D      2$        ADD     R0,R0,R1   ; R0 contains argument x
0E                LD      R4,BASE
0F                JSRR    R4
10                ST      R0,HYPOT
11                BRnzp   NEXT_TASK
12      SQUARE    ADD     R2,R0,#0
13                ADD     R3,R0,#0
14      AGAIN     ADD     R2,R2,#-1
15                BRz     DONE
16                ADD     R0,R0,R3
17                BRnzp   AGAIN
18      DONE      RET
19      BASE      .FILL   SQRT
1A      SIDE1     .BLKW   1
1B      SIDE2     .BLKW   1
1C      HYPOT     .BLKW   1
1D                ...
1E                ...
```

Figure 9.12 The program fragment of Figure 9.10, using a library routine

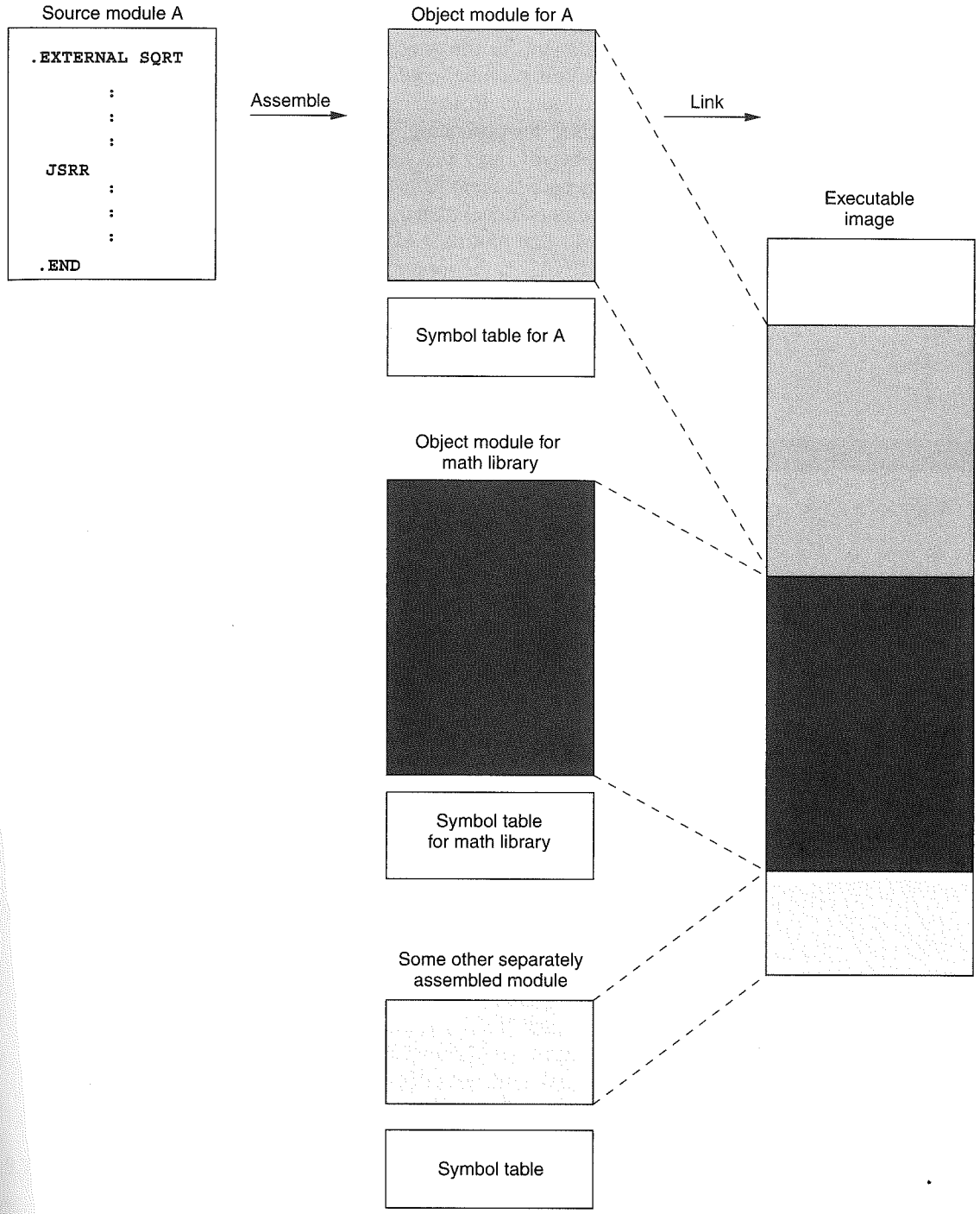


Figure 9.13 An executable image constructed from multiple files

Most application software requires library routines from various libraries. It would be very inefficient for the typical programmer to produce all of them—assuming the typical programmer could produce such routines in the first place. We have mentioned routines from the Math Library. There are also a number of preprocessing routines for producing "pretty" graphic images. There are other routines for a number of other tasks where it would make no sense at all to have the programmer write them from scratch. It is much easier to require only (1) appropriate documentation so that the interface between the library routine and the program that calls that routine is clear, and (2) the use of the proper pseudo-ops such as .EXTERNAL in the source program. The linker can then produce an executable image at link time from the separately assembled modules.

Exercises

9.1 Name some of the advantages of doing I/O through a TRAP routine instead of writing the routine yourself each time you would like your program to perform I/O.

9.2 *a.* How many trap service routines can be implemented in the LC-3? Why?

b. Why must a RET instruction be used to return from a TRAP routine? Why won't a BR (Unconditional Branch) instruction work instead?

c. How many accesses to memory are made during the processing of a TRAP instruction? Assume the TRAP is already in the IR.

9.3 Refer to Figure 9.6, the HALT service routine.

a. What starts the clock after the machine is HALTed? Hint: How can the HALT service routine return after bit [15] of the machine control register is cleared?

b. Which instruction actually halts the machine?

c. What is the first instruction executed when the machine is started again?

d. Where will the RET of the HALT routine return to?

9.4 Consider the following LC-3 assembly language program:

```
        .ORIG   x3000
L1      LEA     R1, L1
        AND     R2, R2, x0
        ADD     R2, R2, x2
        LD      R3, P1
L2      LDR     R0, R1, xC
        OUT
        ADD     R3, R3, #-1
        BRz     GLUE
        ADD     R1, R1, R2
        BR      L2
GLUE    HALT
P1      .FILL   xB
        .STRINGZ "HBoeoakteSmtHaotren!s"
        .END
```

a. After this program is assembled and loaded, what binary pattern is stored in memory location x3005?
b. Which instruction (provide a memory address) is executed after instruction x3005 is executed?
c. Which instruction (provide a memory address) is executed prior to instruction x3006?
d. What is the output of this program?

9.5 The following LC-3 program is assembled and then executed. There are no assemble time or run-time errors. What is the output of this program? Assume all registers are initialized to 0 before the program executes.

```
        .ORIG    x3000
        ST       R0, x3007
        LEA      R0, LABEL
        TRAP     x22
        TRAP     x25
LABEL   .STRINGZ "FUNKY"
LABEL2  .STRINGZ "HELLO WORLD"
        .END
```

9.6 The correct operation of the program in Example 9.1 assumes that the person sitting at the keyboard only types capital letters and the value 7. What if the person types a $? A better program would be one that tests the character typed to be sure it really is a capital letter from among the 26 capital letters in the alphabet, and if it is not, takes corrective action. Your job: Augment the program of Example 9.1 to add a test for bad data. That is, write a program that will type the lowercase representation of any capital letter typed and will terminate if anything other than a capital letter is typed.

9.7 Two students wrote interrupt service routines for an assignment. Both service routines did exactly the same work, but the first student accidentally used RET at the end of his routine, while the second student correctly used RTI. There are three errors that arose in the first student's program due to his mistake. Describe any two of them.

9.8 Assume that an integer greater than 2 and less than 32,768 is deposited in memory location A by another module before the program below is executed.

```
        .ORIG   x3000
        AND     R4, R4, #0
        LD      R0, A
        NOT     R5, R0
        ADD     R5, R5, #2
        ADD     R1, R4, #2
        ;
REMOD   JSR     MOD
        BRz     STORE0
        ;
        ADD     R7, R1, R5
        BRz     STORE1
        ADD     R1, R1, #1
        BR      REMOD
        ;
STORE1  ADD     R4, R4, #1
STORE0  ST      R4, RESULT
        TRAP    x25
        ;
MOD     ADD     R2, R0, #0
        NOT     R3, R1
        ADD     R3, R3, #1
DEC     ADD     R2, R2, R3
        BRp     DEC
        RET
        ;
A       .BLKW 1
RESULT  .BLKW 1
        .END
```

In 20 words or fewer, what does the above program do?

9.9 Recall the machine busy example. Suppose the bit pattern indicating which machines are busy and which are free is stored in memory location x4001. Write subroutines that do the following.

a. Check if no machines are busy, and return 1 if none are busy.
b. Check if all machines are busy, and return 1 if all are busy.
c. Check how many machines are busy, and return the number of busy machines.
d. Check how many machines are free, and return the number of free machines.
e. Check if a certain machine number, passed as an argument in R5, is busy, and return 1 if that machine is busy.
f. Return the number of a machine that is not busy.

9.10 The starting address of the trap routine is stored at the address specified in the TRAP instruction. Why isn't the first instruction of the trap routine stored at that address instead? Assume each trap service routine requires at most 16 instructions. Modify the semantics of the LC-3 TRAP instruction so that the trap vector provides the starting address of the service routine.

9.11 Following is part of a program that was fed to the LC-3 assembler. The program is supposed to read a series of input lines from the console into a buffer, search for a particular character, and output the number of times that character occurs in the text. The input text is terminated by an EOT and is guaranteed to be no more than 1,000 characters in length. After the text has been input, the program reads the character to count.

The subroutine labeled COUNT that actually does the counting was written by another person and is located at address x3500. When called, the subroutine expects the address of the buffer to be in R5 and the address of the character to count to be in R6. The buffer should have a NULL to mark the end of the text. It returns the count in R6.

The OUTPUT subroutine that converts the binary count to ASCII digits and displays them was also written by another person and is at address x3600. It expects the number to print to be in R6.

Here is the code that reads the input and calls COUNT:

```
        .ORIG   x3000
        LEA     R1, BUFFER
G_TEXT  TRAP    x20             ; Get input text
        ADD     R2, R0, x-4
        BRz     G_CHAR
        STR     R0, R1, #0
        ADD     R1, R1, #1
        BRz     G_TEXT
G_CHAR  STR     R2, R1, #0      ; x0000 terminates buffer
        TRAP    x20             ; Get character to count
        ST      R0, S_CHAR
        LEA     R5, BUFFER
        LEA     R6, S_CHAR
        LD      R4, CADDR
        JSRR    R4              ; Count character
        LD      R4. OADDR
        JSRR    R4              ; Convert R6 and display
        TRAP    x25
CADDR   .FILL   x3500           ; Address of COUNT
OADDR   .FILL   x3600           ; Address of OUTPUT
BUFFER  .BLKW   1001
S_CHAR  .FILL   x0000
        .END
```

There is a problem with this code. What is it, and how might it be fixed? (The problem is *not* that the code for `COUNT` and `OUTPUT` is missing.)

9.12 Consider the following LC-3 assembly language program:

```
        .ORIG   x3000
        LEA     R0,DATA
        AND     R1,R1,#0
        ADD     R1,R1,#9
LOOP1   ADD     R2,R0,#0
        ADD     R3,R1,#0
LOOP2   JSR     SUB1
        ADD     R4,R4,#0
        BRzp    LABEL
        JSR     SUB2
LABEL   ADD     R2,R2,#1
        ADD     R3,R3,#-1
        BRP     LOOP2
        ADD     R1,R1,#-1
        BRp     LOOP1
        HALT
DATA    .BLKW   10 x0000
SUB1    LDR     R5,R2,#0
        NOT     R5,R5
        ADD     R5,R5,#1
        LDR     R6,R2,#1
        ADD     R4,R5,R6
        RET
SUB2    LDR     R4,R2,#0
        LDR     R5,R2,#1
        STR     R4,R2,#1
        STR     R5,R2,#0
        RET
        .END
```

Assuming that the memory locations at DATA get filled in before the program executes, what is the relationship between the final values at DATA and the initial values at DATA?

9.13 The following program is supposed to print the number 5 on the screen. It does not work. Why? Answer in no more than ten words, please.

```
        .ORIG   x3000
        JSR     A
        OUT
        BRnzp   DONE
A       AND     R0,R0,#0
        ADD     R0,R0,#5
        JSR     B
        RET
DONE    HALT
ASCII   .FILL   x0030
B       LD      R1,ASCII
        ADD     R0,R0,R1
        RET
        .END
```

9.14 Figure 9.6 shows a service routine to stop the computer by clearing the RUN latch, bit [15] of the Machine Control Register. The latch is cleared by the instruction in line 14, and the computer stops. What purpose is served by the instructions on lines 19 through 1C?

9.15 Suppose we define a new service routine starting at memory location x4000. This routine reads in a character and echoes it to the screen. Suppose memory location x0072 contains the value x4000. The service routine is shown below.

```
            .ORIG x4000
            ST R7, SaveR7
            GETC
            OUT
            LD R7, SaveR7
            RET
SaveR7      .FILL x0000
```

a. Identify the instruction that will invoke this routine.
b. Will this service routine work? Explain.

9.16 The two code sequences *a* and *b* are assembled separately. There is one error that will be caught at assemble time or at link time. Identify and describe why the bug will cause an error, and whether it will be detected at assemble time or link time.

a.

```
            .ORIG x3200
SQRT        ADD     R0, R0, #0
            ; code to perform square
            ; root function and
            ; return the result in R0
            RET
            .END
```

b.

```
            .EXTERNAL SQRT
            .ORIG   x3000
            LD      R0,VALUE
            JSR     SQRT
            ST      R0,DEST
            HALT
VALUE       .FILL   x30000
DEST        .FILL   x0025
            .END
```

9.17 Shown below is a partially constructed program. The program asks the user his/her name and stores the sentence "Hello, name" as a string starting from the memory location indicated by the symbol HELLO. The program then outputs that sentence to the screen. The program assumes that the user has finished entering his/her name when he/she presses the Enter key, whose ASCII code is x0A. The name is restricted to be not more than 25 characters.

Assuming that the user enters Onur followed by a carriage return when prompted to enter his/her name, the output of the program looks exactly like:

```
Please enter your name: Onur
Hello, Onur
```

Insert instructions at (a)–(d) that will complete the program.

```
            .ORIG x3000
            LEA   R1,HELLO
AGAIN       LDR   R2,R1,#0
            BRz   NEXT
            ADD   R1,R1,#1
            BR    AGAIN
NEXT        LEA   R0,PROMPT
            TRAP  x22              ; PUTS
            ------------ (a)
AGAIN2      TRAP  x20              ; GETC
            TRAP  x21              ; OUT
            ADD   R2,R0,R3
            BRz   CONT
            ------------ (b)
            ------------ (c)
            BR    AGAIN2
CONT        AND   R2,R2,#0
            ------------ (d)
            LEA   R0, HELLO
            TRAP  x22              ; PUTS
            TRAP  x25              ; HALT
NEGENTER    .FILL xFFF6            ; -x0A
PROMPT      .STRINGZ "Please enter your name: "
HELLO       .STRINGZ "Hello, "
            .BLKW #25
            .END
```

9.18 The program below, when complete, should print the following to the monitor:

ABCFGH

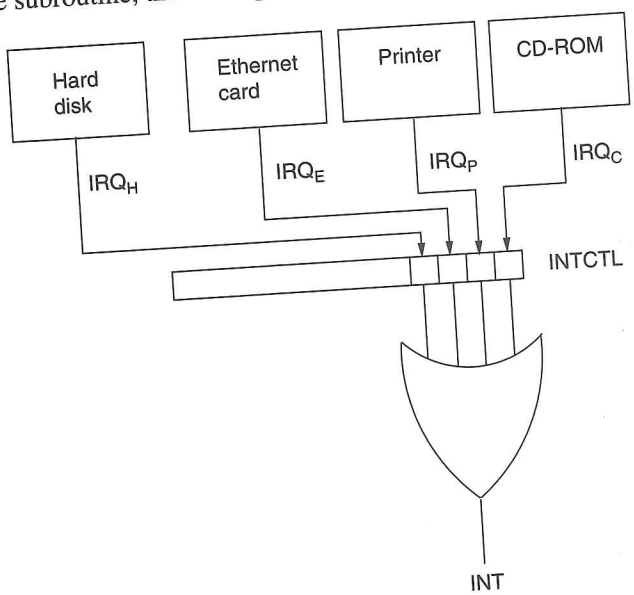
Insert instructions at (a)–(d) that will complete the program.

```
            .ORIG x3000
            LEA   R1, TESTOUT
BACK_1      LDR   R0, R1, #0
            BRz   NEXT_1
            TRAP  x21
            ------------ (a)
            BRnzp BACK_1
            ;
NEXT_1      LEA   R1, TESTOUT
BACK_2      LDR   R0, R1, #0
            BRz   NEXT_2
            JSR   SUB_1
            ADD   R1, R1, #1
            BRnzp BACK_2
            ;
NEXT_2      ------------ (b)
            ;
SUB_1       ------------ (c)

K           LDI   R2, DSR
            ------------ (d)

            STI   R0, DDR
            RET
DSR         .FILL xFE04
DDR         .FILL xFE06
TESTOUT     .STRINGZ "ABC"
            .END
```

9.19 A local company has decided to build a real LC-3 computer. In order to make the computer work in a network, four interrupt-driven I/O devices are connected. To request service, a device asserts its interrupt request signal (IRQ). This causes a bit to get set in a special LC-3 memory-mapped interrupt control register called INTCTL which is mapped to address xFF00. The INTCTL register is shown below. When a device requests service, the INT signal in the LC-3 data path is asserted. The LC-3 interrupt service routine determines which device has requested service and calls the appropriate subroutine for that device. If more than one device asserts its IRQ signal at the same time, only the subroutine for the highest priority device is executed. During execution of the subroutine, the corresponding bit in INTCTL is cleared.



The following labels are used to identify the first instruction of each device subroutine:

HARDDISK ETHERNET PRINTER CDROM

For example, if the highest priority device requesting service is the printer, the interrupt service routine will call the printer subroutine with the following instruction:

JSR PRINTER

Finish the code in the LC-3 interrupt service routine for the following priority scheme by filling in the spaces labeled (a)–(k). The lower the number, the higher the priority of the device.

1. Hard disk
2. Ethernet card
3. Printer
4. CD-ROM

```
            LDI   R1, INTCTL
DEV0        LD    R2, ------ (a)
            AND   R2, R2, R1
            BRnz  DEV1
            JSR   ---------- (b)
            ---------------- (c)
    ;
DEV1        LD    R2, ------ (d)
            AND   R2, R2,  R1
            BRnz  DEV2
            JSR   ---------- (e)
            ---------------- (f)
    ;
DEV2        LD    R2, ------ (g)
            AND   R2, R2, R1
            BRnz  DEV3
            JSR   ---------- (h)
            ---------------- (i)
    ;
DEV3        JSR  ----------- (j)
    ;
END         ---------------- (k)

INTCTL      .FILL   xFF00
MASK8       .FILL   x0008
MASK4       .FILL   x0004
MASK2       .FILL   x0002
MASK1       .FILL   x0001
```